CONTROLLO DI UN BRACCIO ROBOTICO TRAMITE APPRENDIMENTO PER RINFORZO

Oliviero Nardi

Università degli Studi di Verona
Corso di Laurea in Informatica

Luglio 2019

Abstract

Quello dell'apprendimento automatico è un campo che negli ultimi anni ha conosciuto un forte sviluppo, mostrando risultati interessanti in una varietà di applicazioni differenti. In particolare, l'apprendimento per rinforzo può essere utilizzato in robotica come strumento per lo sviluppo di controllori generalizzabili capaci di adattarsi e rispondere alla situazione in cui si trovano; questo è un grande vantaggio rispetto ai controllori classici. Tuttavia, questa tecnica non è esente da difetti: ad esempio, nel problema della generazione delle traiettorie, i controllori basati sull'apprendimento non riescono sempre a portare a distanza millimetrica l'end-effector del braccio rispetto al target. In questo progetto si è esaminato un possibile approccio misto al problema della generazione delle traiettorie di un braccio robotico a 7 gradi di libertà in presenza di un ostacolo, che sfrutta in combinazione tecniche basate sull'apprendimento per rinforzo e metodi di controllo classici. Sono state studiate due varianti: una in cui il controllore classico si occupa solamente di generare l'ultima parte della traiettoria ed una in cui il controllore aiuta anche ad evitare l'ostacolo, filtrando azioni che comporterebbero una collisione.

Contents

Introduzione

Deep-Q-Learning

Iniziamo presentando brevemente e sinteticamente DQN, l'algoritmo di apprendimento utilizzato.

Apprendimento per rinforzo

Per "apprendimento per rinforzo" si intende una classe di algoritmi di apprendimento automatico di tipo *goal-oriented*. L'idea di fondo è quella di utilizzare un sistema basato su ricompense (generalmente valori numerici) per gestire l'apprendimento da parte di un agente di un determinato tipo di comportamento mirato a massimizzare le ricompense stesse ([6]). Il fondamento dell'apprendimento per rinforzo è l'interazione: un *agente* interagisce con l'*ambiente* che si trova in un certo *stato* e deve compiere delle *azioni*, ricevendo delle *ricompense*. Un formalismo che ci permette di modellare situazioni simili è quello dei *Markov Decision Processes* (MDP); essi forniscono un linguaggio matematico utile a modellare situazioni in cui è necessario prendere decisioni a fronte di incertezza. Un MDP è formato da una quintula $\langle S, A, P, R, \lambda \rangle$:

- S: insieme degli stati
- A: insieme delle azioni
- P: dinamiche di transizioni, dove $P(s'|s, a)$ denota la probabilità di raggiungere lo stato prossimo s' dallo stato s compiendo l'azione a
- R: insieme delle ricompense
- γ: *discount factor*, nel range $[0, 1]$.

In particolare, in robotica, consideriamo MDP *episodici*, dove esiste uno stato terminale che conclude l'episodio (e nel nostro caso, un *timeout* T dopo il quale l'episodio viene concluso in ogni caso). Per altri dettagli riguardo agli MDP e in particolare al loro utilizzo nell'apprendimento per rinforzo in robotica, si veda [9].

Ricapitolando, nell'apprendimento per rinforzo il nostro agente deve imparare tramite l'interazione con l'ambiente un certo tipo di comportamento (atto a massimizzare la ricompensa). Si distingue quindi dall'apprendimento supervisionato classico poiché non è necessario fornire in anticipo nessun insieme di dati input/output etichettati; questo è desiderabile in casi come il nostro, data la difficoltà di fornire esempi di traiettorie etichettate.

Algoritmi *model-based* ed algoritmi *model-free*

Nell'apprendimento per rinforzo, per *modello* si intende qualsiasi cosa che permetta ad un agente di prevedere come l'ambiente risponderà alle sue azioni: dato uno stato ed un'azione, il modello costruisce una previsione dello stato prossimo e la ricompensa ([7]). Negli algoritmi *model-based*, durante l'apprendimento l'agente costruisce esplicitamente un modello; dopodiché, pianifica direttamente la sequenza di azioni da compiere mediante una ricerca sul modello stesso. Contrariamente, negli algoritmi *model-free*, l'agente non costruisce un modello, ed usa l'esperienza per migliorare la *policy* (politica di scelta delle azioni) o la *value-function* (stima della ricompensa futura). L'algoritmo che adesso andremo ad introdurre, Q-Learning, è un classico esempio di algoritmo *model-free*.

Q-Learning

Q-Learning è un algoritmo di apprendimento per rinforzo. Un agente deve apprendere una cosiddetta Q-function che dà l'utilità prevista di scegliere una certa azione in uno spazio di stati, cercando di approssimare $Q*$, la funzione di *action-value* ottima. Denotiamo quindi $Q(s, a)$ come il valore di scegliere l'azione a nello stato s; l'obiettivo è quello di aggiornare la Q-function grazie all'esperienza acquisita dall'agente per avvicinarsi a $Q*$. Per fare questo, si usa la seguente equazione di aggiornamento ([8]):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{1}$$

Dove α è il learning-rate, r_t è la ricompensa registrata all'istante t, γ è il discount factor (per ammortizzare il valore delle ricompense future sulle scelte dell'agente nell'immediato) e $max_a\, Q(s_{t+1}, a)$ è una stima delle ricompense future. L'idea è quella di far percorrere all'agente un certo numero di *episodi* (dipendenti dal task che stiamo considerando) per raccogliere dati sugli stati, le azioni e le ricompense ad essi associati; dopo ogni episodio la Q-function viene aggiornata sulla base dell'equazione sopra presentata. Q-learning è detto *off-policy*, perché l'apprendimento non interviene sulla politica di scelta delle azioni, che rimane fissa, ad esempio ϵ-greedy: con probabilità ϵ viene scelta un'azione casuale, altrimenti si sceglie l'azione che massimizza $Q(s, a)$. Generalmente ϵ viene fatto decrescere (questo permette di esplorare inizialmente lo spazio delle azioni). Si veda [8] per ulteriori dettagli e per l'algoritmo in dettaglio.

DQN

Nel caso di DQN (Deep-Q-Network), l'agente che impara a mappare coppie di stato-azione alle ricompense è una rete neurale, un approssimatore di funzioni universale. Questo è utile quando le combinazioni stato-azione sono troppe per poter computare ed aggiornare agevolmente la Q-function: la funzione che la rete neurale deve approssimare è quindi $Q*$ ([9]). In termini più concreti, dato uno stato, ci aspettiamo che la rete neurale predica l'azione che dobbiamo scegliere

se vogliamo massimizzare la ricompensa. In questo caso, l'input della rete è costituito solamente dallo stato (e non dalla coppia stato-azione) e l'output è una stima del Q-value di ogni possibile azione (**figura 1**). Notiamo quindi che $max_a\, Q(s_{t+1}, a)$ è calcolato tramite la rete neurale. L'apprendimento verrà quindi fatto secondo l'algoritmo backpropagation per la seguente funzione di costo (si veda [3] per l'algoritmo completo):

$$L(\theta_t) = (r_t + \gamma \max_a Q(s_{t+1}, a, \theta_t) - Q(s_t, a_t, \theta_t))^2 \tag{2}$$

Dove θ_t sono i parametri della rete all'istante t.

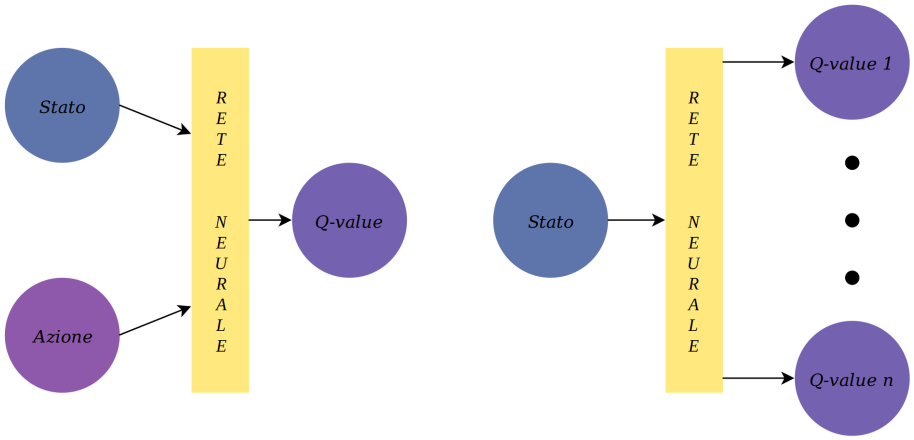

Figure 1: Sinistra: Architettura naive di un Q-network. Destra: Approccio ottimizzato utilizzato da [3]

Presentazione del problema

Motivazioni

L'apprendimento per rinforzo offre applicazioni molto interessanti nella robotica, specialmente nello sviluppo di controllori sofisticati. Un importante vantaggio è l'adattabilità: mentre i controllori basati su cinematica sono fortemente *task-dependent* (ovvero richiedono una soluzione ad-hoc specifica per ogni problema), i controllori appresi offrono una certa plasticità: una variazione nell'ambiente o nel task (e.g. guasto di una giuntura) non implica un dover riscrivere da capo l'intero sistema, bensì semplicemente un riapprendimento che, come rielvato in questo progetto, è anche relativamente breve (6-7 ore). Oltre a questo, tramite l'apprendimento per rinforzo possiamo lasciare che il robot apprenda autonomamente la strategia ottimale, senza doverci preoccupare di ingegnerizzare esplicitamente una soluzione: questo è particolarmente utile in problemi difficili da gestire analiticamente, come generare una traiettoria evitando un ostacolo.

Ciònonostante, ci sono degli aspetti negativi da considerare: spesso, i controllori sviluppati tramite apprendimento sono meno precisi di quelli basati su cinematica. Quindi, possiamo notare come gli approcci classici siano precisi, ma molto rigidi, mentre gli approcci con apprendimento siano flessibili ma imprecisi. In questo progetto si è provato a risolvere questa dicotomia cercando di combinare l'approccio classico a quello basato sull'apprendimento: si è dimostrato come sia possibile sviluppare facilmente un controllore capace di evitare un ostacolo tramite DQN, e come sia possibile migliorarlo tramite l'innesto di un controllore cinematico sulla fase finale. In questo modo, si sfrutta sia l'apprendimento per rinforzo per sviluppare rapidamente un controllore capace di evitare ostacoli, sia la precisione della cinematica per ottenere una distanza millimetrica. Si è inoltre analizzata una variante che sfrutta un controllore aggiuntivo per migliorare sensibilmente il success rate.

Generazione delle traiettorie con ostacolo

Nel progetto, il problema considerato è quello della generazione di traiettorie per un braccio robotico a 7 gradi di libertà in presenza di un ostacolo. Le traiettorie consistono in una serie di **joint states** (angolazioni delle giunture) che il braccio deve assumere per spostarsi dalla configurazione iniziale a quella finale. L'obiettivo è quello di raggiungere un target a distanza millimetrica evitando un ostacolo. Il target viene generato casualmente in un punto del workspace del braccio, ed è quindi sempre raggiunbile. L'ostacolo è generato casualmente in un punto che giace in un cilindro di raggio 2 cm tra la posizione iniziale dell'end-effector ed il target (in modo leggermente distanziato dagli estremi, così da non generare ostacoli adiacenti alla posizione iniziale od al target). L'ostacolo ha un raggio casuale tra i 3 e i 6 cm. Oltre a tutto ciò, si è imposto un limite di

500 passi (durante il train) per episodio: se il controllore impiega più di questo numero di step per raggiungere il target, è considerato come un fallimento.

Controllori

Scopo del progetto era quello di studiare la possibilità di utilizzo in congiunta di metodi basati sull'apprendimento per rinforzo e metodi di controllo classici. Per la parte di apprendimento per rinforzo è stato utilizzata una variante neurale del Q-Learning, ovvero l'algoritmo DQN. Per la parte di controllo classico, è stato preferito non utilizzare un controllore a basso livello tra quelli disponibili per il braccio utilizzato (Franka Emika Panda) per non appesantire la fase di training. È stato scriptato un controllore in Python che ha lo scopo di verificare, data la posizione raggiunta e la posizione da raggiungere, che esista almeno una traiettoria che ci porti a distanza millimetrica. Essendo nota la posizione (in termini di **joint states**) del target, esso non fa altro che far convergere a passi di `step` gradi la posizione dell'end-effector fino a quella desiderata.

Sono state proposte due versioni alternative.

1. Nella prima, un controllore basato sull'algoritmo DQN genera la traiettoria che va dalla posizione iniziale fino a 10 cm di distanza dal target; qui, l'esecuzione è ceduta al controllore "classico" che chiude la distanza rimanente, fino a distanza millimetrica.

2. Nella seconda versione, in aggiunta a quanto detto sopra, un controllore secondario verifica ad ogni passo se questo passo implicherebbe una collisione; contemporaneamente, verifica se il braccio si è incagliato in un blocco da cui non sembra uscire. In ambedue i casi, il controllore forza un'azione alternativa a quella scelta dalla rete neurale.

Controllore per la rilevazione delle collisioni

Seguono maggiori dettagli sul controllore per la rilevazione delle collisioni/blocchi.

1. **Rilevazione delle collisioni.** In ogni stato, il controllore esegue la prima azione (relativamente all'ordinamento secondo i punteggi assegnati dalla rete a ciascuna azione) safe (ovvero, che non porti ad una collisione). Ciò significa che se l'azione con punteggio massimo è sicura, si esegue quella; altrimenti, si esegue la seconda, e via dicendo. Se nessuna azione è safe, c'è una collisione.

2. **Rilevazione dei blocchi.** Data la posizione corrente, se non siamo distanti almeno 30 cm da dov'eravamo 14 step prima, allora probabilmente il braccio è bloccato, ovvero, continua a muoversi nello stesso intorno. In questo caso, viene eseguita un'azione casuale che sia safe, in modo da sbloccarsi. Questo controllore non è attivo durante la fase di train (essendoci già una policy ϵ-greedy), tuttavia in fase di testing alza il success rate di circa 7 punti percentuali (da 89.23% a 96.14%).

Braccio robotico: Franka Emika Panda

Il braccio robotico utilizzato in questo progetto è il Panda di Franka Emika (https://www.franka.de/). Si tratta di un braccio robotico industriale a 7 gradi di libertà. Tutte le simulazioni sono state fatte tramite l'API ufficiale offerta da franka emika (https://frankaemika.github.io/docs/libfranka.html). In **figura 2** e **figura 3** sono presentati i dettagli del braccio. Per semplificare il problema, non è stato utilizzato il grasper, ragion per cui sono state utilizzate solamente 5 giunture.

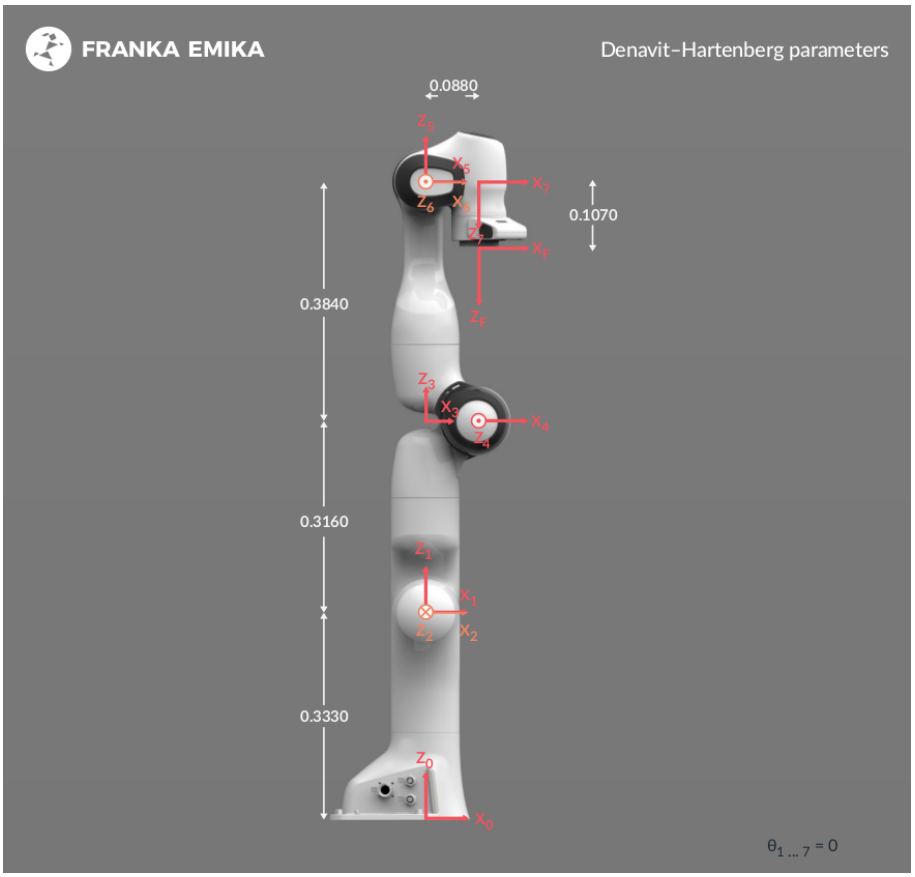

Figure 2: Panda Arm

Modellazione del problema

Rappresentazione dello stato

È stato scelto di non rappresentare gli stati in termini di immagini. Questo ha avuto due vantaggi. In primo luogo, ha permesso di astrarre completamente (nella fase di train) dalle API di franka emika, non dovendo generare in alcun modo una rappresentazione visiva della situazione. In secondo luogo ha permesso di semplificare notevolmente la complessità della rete neurale, passando da una rappresentazione matriciale (con possibilmente centinaia di nodi in input) ad una rappresentazione a 12 nodi di input. Quindi, in essenza, uno stato è rappresentato da 12 valori:

1. I primi 5 valori rappresentano i 5 **joint states** utilizzati nel problema
2. I successivi 3 valori rappresentano le coordinate cartesiane del target

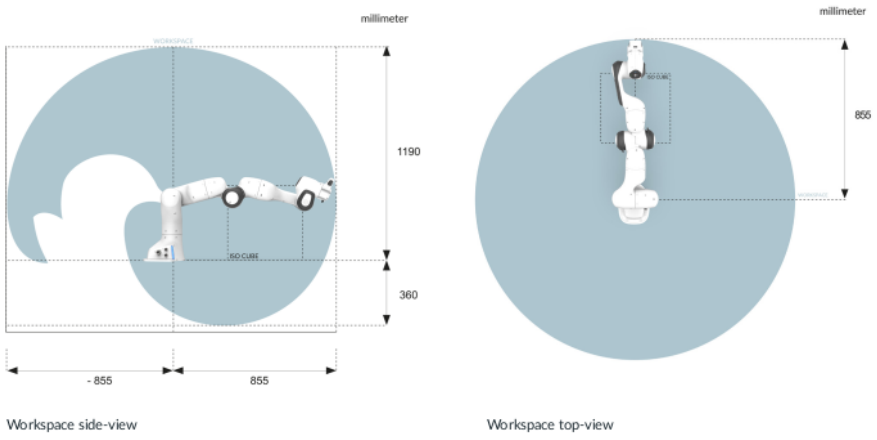

Figure 3: Workspace

3. I successivi 3 valori rappresentano le coordinate cartesiane dell'ostacolo
4. L'ultimo valore rappresenta il raggio dell'ostacolo

Azioni

L'output layer è composto da 10 nodi, ovvero, vi sono 10 possibili azioni che l'agente può eseguire. Ognuna di queste corrisponde ad aumentare o diminuire uno dei possibili 5 giunti (tra quelli utilizzati) del braccio di un certo valore α prefissato.

Rappresentazione delle collisioni

In questo progetto non è mai stato usato il braccio robotico direttamente ma si è sempre lavorato in simulato. Come detto sopra, si è preferito non utilizzare i controllori offerti da libfranka poiché questo avrebbe rallentato di molto la fase di train ed aggiunto ulteriori complicazioni. Conseguentemente, è stato necessario rappresentare in maniera approssimata le collisioni. Per fare ciò, tramite i parametri DH (si veda **figura 2**) è stata ricostruita la catena cinematica del braccio, e da questa è stato possibile rappresentare, in funzione dei **joint states**, il braccio come catena di segmenti (si veda **figura 4**). È stato poi banale passare ad una rappresentazione come catena di cilindri di raggio 2 cm basato sui segmenti precedentemente ricavati. Una collisione si verifica quando l'ostacolo interseca uno qualsiasi di questi cilindri.

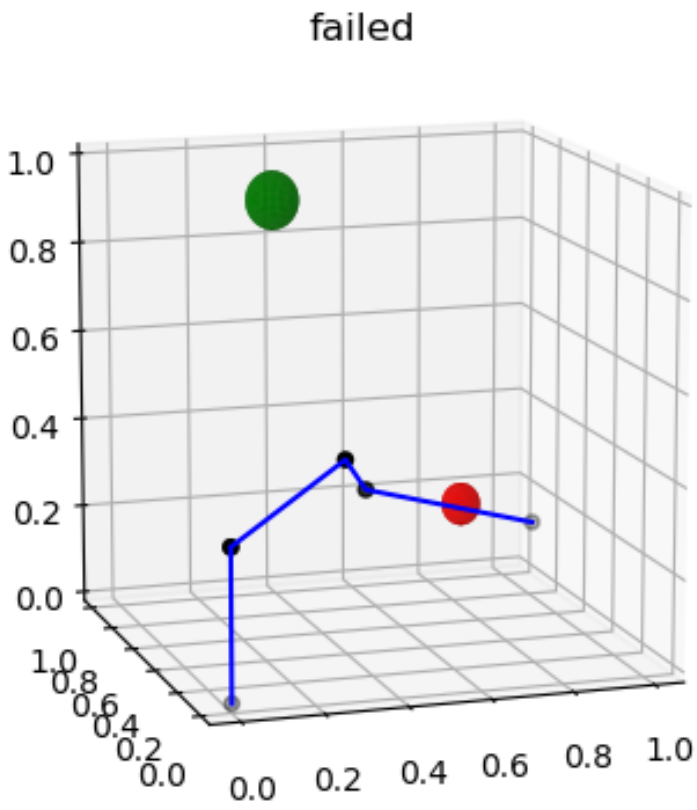

Figure 4: Rappresentazione collisioni con segmenti

Fase di apprendimento

Varianti DQN adottate

Ricompensa sparsa

Un punto cruciale dello sviluppo è quello della modellazione della ricompensa. Una possibile soluzione è quella di adottare un sistema di ricompense continue: ad esempio, basato sulla distanza dal target. Tuttavia questa soluzione è problematica per un braccio a 7 gradi di libertà, vista la sua ridondanza; una posizione vicina al target potrebbe essere pessima dal punto di vista della traiettoria, ovvero può non portare agevolmente a raggiungere il target. Per risolvere questo problema è stato adottato, come proposto in [2], un approccio a ricompensa sparsa, che semplifica notevolmente l'apprendimento. Nella tabella

seguente sono riportati i valori scelti:

-3	In caso di timeout
-2	In caso di collisioni
-1	In caso di fallimento da parte del controllore a basso livello
1	In caso di successo millimetrico
0	Altrimenti (passo intermedio)

Inizialmente il train è stato eseguito su valori di ricompensa negativi tutti pari a -1, ma questa versione otteneva pessimi risultati; dopo uno studio della distribuzione dei fallimenti, rilevando che i timeout erano molto più frequenti degli ostacoli, si è provato a penalizzare maggiormente i timeout, ottenendo un notevole speedup nel train. Si veda la **figura 5**: questa ingegnerizzazione delle ricompense si rivela cruciale.

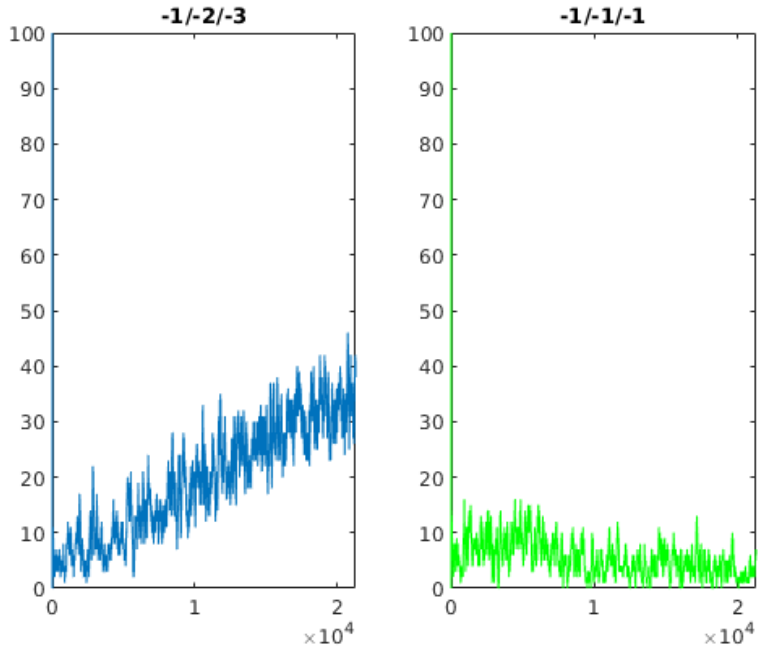

Figure 5:

Priority Experience Replay

Una caratteristica essenziale del DQN è l'*Experience Replay*: l'idea è, al posto che scartare un episodio una volta che esso sia terminato, di memorizzare l'esperienza in una tabella contente transizioni di stati, azioni e ricompense, permettendo di eseguire l'apprendimento su di essa più volte (si veda [3]). Questo permette di velocizzare la fase di apprendimento. In questo progetto, è stata utilizzata la variante *Priority Experience Replay* ([4]). L'idea deriva dal fatto che, adottando lo schema di ricompense sparse sopra proposto, molte coppie stato-azione sono associate alla ricompensa 0; le ricompense significative (negative e positive) sono molto rare. Per questo motivo, piuttosto che campionare casualmente la memoria, vengono selezionate in misura uguale tuple contenenti ricompense nulle (passi intermedi) e tuple contenenti ricompense non-nulle (passi terminali). Questo, come rilevato in [2], ha un impatto cruciale sul success rate.

Adaptive discount factor

Il discount factor (γ), nel Q-learning, è una costante utilizzata per smorzare l'effetto delle ricompense future sul calcolo dei valori delle azioni. Con $\gamma = 0$ ignoreremmo completamente le ricompense future, mentre con $\gamma = 1$ le azioni immediate e quelle future avrebbero lo stesso peso. In questo progetto, piuttosto che selezionare un valore fisso, γ è stato scalato dinamicamente: inializzato a 0.40, viene cresce di un fattore del 100.0125% ad ogni iterazione fino ad un massimo di 0.99. Questo approccio ha migliorato notevolmente il tempo di train.

ϵ-greedy policy

Per esplorare lo state-space è stata impiegata una politica ϵ-greedy ([3]). Secondo questo approccio, ad ogni iterazione l'agente ha una possibilità ϵ di scegliere un'azione completamente casuale. ϵ è inizializzato ad 1.00 ed il suo valore è diminuito di un fattore del 99.95% ad ogni episodio, fino ad un minimo di 0.01. Così facendo, l'agente ha la possibilità di esplorare una fetta relativamente vasta di coppie stato-azione. Una volta accumulata una certa esperienza dell'ambiente, può iniziare ad usare la rete per selezionare le azioni più adeguate.

Double DQN (DDQN)

Un problema del DQN standard è che la rete utilizza l'operatore *max* sia per stimare il valore di ciascuna azione sia per selezionare l'azione migliore. Questo porta ad un problema di sopravvalutazione, in cui le azioni sopravvalutate finiscono sempre sopra a quelle sottovalutate (per una dimostrazione formale, si veda [10]). Possiamo vedere questo problema come uno "sparare ad un bersaglio in movimento", in cui utilizziamo la rete per valutare le azioni mentre allo stesso tempo la aggiorniamo. Per ovviare a questo problema è possibile adottare DDQN, una variante di DQN in cui viene disaccopiata la valutazione delle

azioni dalla loro selezione (basato su Double Q-Learning, [1]). In questo algoritmo, una seconda rete topologicamente identica alla prima (ed inizializzata agli stessi parametri) viene utilizzata. La rete principale (model) è impiegata ancora per selezionare l'azione da eseguire, mentre una "rete obiettivo" (target model) è impiegata per calcolare il Q-value obiettivo relativo all'eseguire quell'azione nello stato prossimo. La fase di apprendimento rimane uguale alla precedente, aggiornando i parametri del modello principale ad ogni episodio; in aggiunta, ogni 2000 episodi i pesi del target model vengono aggiornati (ovvero, eseguiamo $\theta' \leftarrow \theta$). La funzione di costo diviene quindi (riadattato da [5]):

$$L(\theta_t) = (r_t + \gamma Q(s_{t+1}, \underset{a}{argmax}\, Q(s_{t+1}, a, \theta_t), \theta'_t) - Q(s_t, a_t, \theta_t))^2 \qquad (3)$$

Dove θ e θ' rappresentano rispettivamente i pesi del model e del target model. Notiamo che la funzione max è scomparsa.

Architettura della rete ed implementazione

Evitando di usare immagini come input per la nostra rete, abbiamo potuto mantenere molto più ridotte le sue dimensioni. Per selezionare l'architettura della rete, in termini di topologia, è stato semplicemente fatto un confronto di performance su diverse architetture per brevi training. Tra quelle testate, l'architettura che è emersa come migliore è stata quella formata da due hidden layers da 128 nodi ciascuno.

Per implementare la rete neurale e gli algoritmi di apprendimento (backpropagation) si è utilizzata la libreria Keras (http://keras.io/) con TensorFlow.

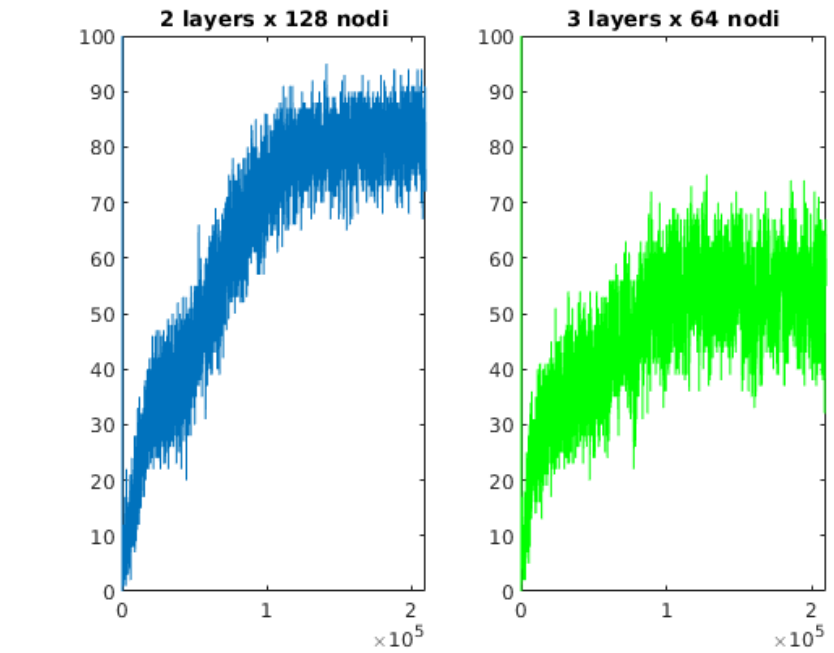

Figure 6: Confronto con Architettura a 3 layer, 64 nodi ciascuno

Inserimento dei controllori nella fase di train

Il punto cruciale di questo progetto è l'inserimento dei controllori nella fase di train. Come spiegato in precedenza, si è voluto testare due diverse implementazioni, con due controllori diversi. In entrambi i casi, si è cercato di rendere il rapporto tra il train e il testing il più omogeneo possibile, ovvero lasciando i rispettivi controllori attivi durante la fase di apprendimento.

Per quanto riguarda il primo controllore, ovvero quello che conclude la traiettoria, la situazione era relativamente semplice. Data una nuova posizione iniziale, la rete genera la traiettoria che porta l'end-effector dal punto iniziale fino ad una distanza di 10 cm dal target. Se durante questa traiettoria vi è una collisione o si va in timeout, la rete registra una ricompensa negativa. Altrimenti, passa l'esecuzione al controllore a basso livello, che se possibile, genererà una traiettoria dal punto in cui si trova fino ad una distanza millimetrica dal target. Solo una volta che questa fase è conclusa la rete riceverà la ricompensa positiva: se vi è una collisione o questo non avviene entro un certo numero di step, la rete registra una ricompensa negativa.

Il secondo controllore, descritto nella sezione **Controllore per la rilevazione**

delle collisioni è impiegato in una versione separata del progetto. Anch'esso è lasciato attivo durante la fase di train: per sfruttarlo durante l'apprendimento, al posto che passare all'ambiente l'azione selezionata dalla rete, viene passata una lista di azioni ordinate secondo il Q-value. A questo punto il controllore eseguirà l'azione adeguata (secondo quanto descritto sopra, nella specifica del controllore) e ritornerà, insieme allo stato prossimo e le altre informazioni, l'azione realmente eseguita. Questa azione verrà quindi inserita in una tupla per fare *Experience Replay* ed aggiornare i parametri della rete. Da qua in poi non ci sono altre variazioni salienti rispetto all'algoritmo standard.

Dati sull'apprendimento

Versione con controllore semplice

In **figura 7** è mostrata l'evoluzione del success rate della versione con controllore semplice. Dopo i primi 80000 step (nota bene: singoli step, non episodi), la distribuzione dei fallimenti era più o meno pari (52% per collisioni e 48% timeouts); alla fine, il 74% dei fallimenti erano per collisioni. Si noti che la percentuale di fallimenti dovuta al controllore a basso livello (ovvero, il numero di casi in cui è stata raggiunta una distanza di 10 cm dal target ma il controllore non è arrivato a distanza millimetrica) è piuttosto bassa: 1376 casi su 209502 episodi. Il train ha impiegato un tempo abbastanza breve (circa 6-7 ore) su un pc di fascia medio-bassa.

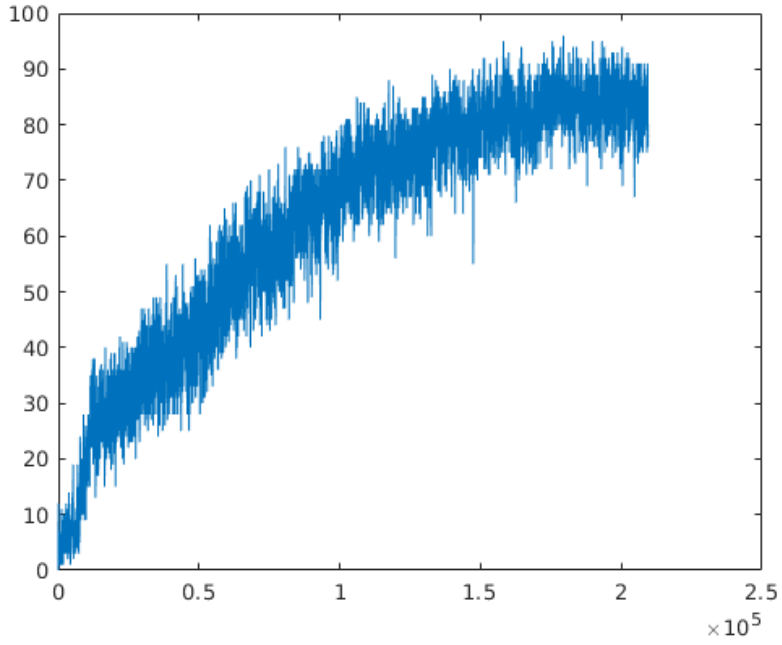

Figure 7: success rate versione semplice

Versione con rilevamento di collisioni

In **figura 8** è mostrata l'evoluzione del success rate della versione con rilevamento delle collisioni. Si noti che l'apprendimento è molto più repentino della versione precedente. Per quanto riguarda la distribuzione dei fallimenti, il pattern sembra inverso al caso precedente: dopo i primi 80000 step, l'85% fallimenti erano collisioni, mentre alla fine, solo il 59% dei fallimenti erano collisioni ed il 40% erano timeout. Il discorso per i fallimenti dovuti al controllore a basso livello è analogo al caso precedente: 2005 casi su 214302 episodi. Il train ha impiegato un tempo abbastanza breve (circa 6-7 ore) su un pc di fascia medio-bassa.

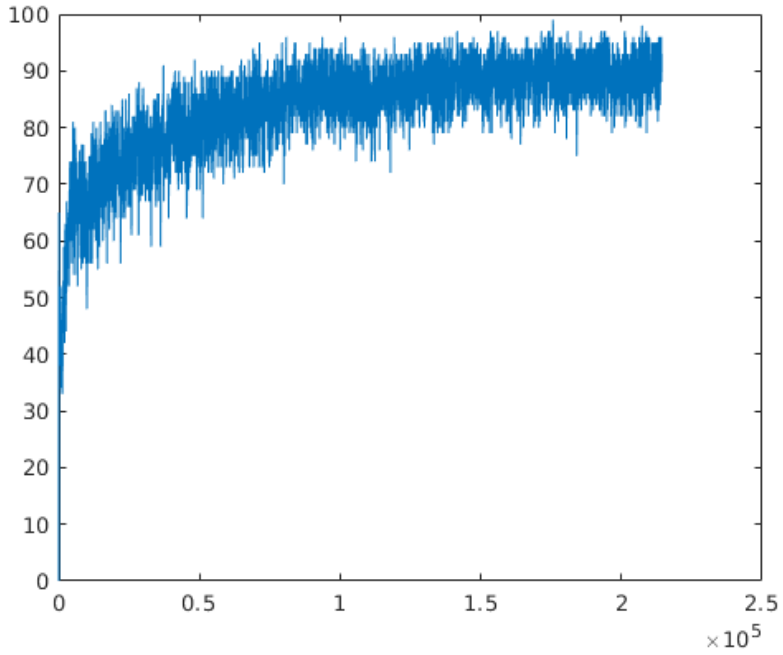

Figure 8: success rate versione rilevamento delle collisioni

Esperimenti e conclusioni

Versione con controllore semplice

Dopo la fase di train, è stato eseguito un test su 10000 episodi per verificare la performance del modello. L'ambiente è rimasto pressocché invariato, se non per una variazione nello step size (ridotto da 5° a 2.5°), una riduzione del timeout (ridotto a 150 steps) e la rimozione dell'azione casuale. Inoltre, gli episodi partono sempre dalla configurazione standard. Con queste caratteristiche, sono state raccolte le seguenti statistiche:

- Success rate **millimetrico** (< 1 mm) pari al 90.58%
- Dei rimanenti episodi, 5.32% sono collisioni e 4.10% timeouts
- Una media di 46.62 step eseguiti per episodio, compresa la fase finale
- Nel 99.51% dei casi in cui la rete ci ha portato a 10 cm di distanza abbiamo avuto un successo

Versione con rilevamento di collisioni

Dopo la fase di train, è stato eseguito un test su 10000 episodi per verificare la performance del modello. Valgono le stesse modifiche del caso precedente: inoltre, è stato attivato il rilevamento dei blocchi (si veda sezione **Controllore per la rilevazione delle collisioni**). Con queste caratteristiche, sono state raccolte le seguenti statistiche:

- Success rate **millimetrico** (< 1 mm) pari al 96.14%
- Dei rimanenti episodi, 2.88% sono collisioni e 0.98% timeouts
- Una media di 23.41 step eseguiti per episodio, compresa la fase finale
- Nel 99.43% dei casi in cui la rete ci ha portato a 10 cm di distanza abbiamo avuto un successo

Conclusioni e possibili lavori ulteriori

Come visto nelle sezioni precedenti, l'introduzione di un controllore a basso livello permette di risolvere uno dei problemi dell'apprendimento per rinforzo nel controllo dei bracci robotici, andando a raggiungere con precisione millimetrica il target. Questo permette di sfruttare la flessibilità degli approcci neurali in combinazione con la precisione e robustezza di algoritmi classici. Inoltre, è stato mostrato come l'introduzione di sistemi di controllo per la rilevazione delle collisioni potrebbe migliorare drasticamente il success rate. Infine, è da sottolineare che il train di questi sistemi ha impiegato un tempo abbastanza rapido (6-7 ore) su un pc di fascia medio-bassa, confermando come questi controllori offrano una capacità di adattarsi impiegabile in casi reali.

Possibili lavori ulteriori potrebbero riguardare i seguenti problemi:

1. **Generalizzazione del problema ad n ostacoli.** In questo progetto, è stata considerata solo una casistica limitata (un singolo ostacolo sempre circa a metà tra end-effector e target). Una possibile miglioria sarebbe quella di generalizzare ad un qualsiasi numero di ostacoli. Una possibile soluzione, benché computazionalmente difficile, sarebbe quella di rappresentare lo state-space tramite matrici di occupazione; alternativamente si potrebbero cercare altre rappresentazioni compatte di gruppi di ostacoli.
2. **Analisi del problema utilizzando controllori reali del braccio Panda.** In questo progetto sono stati utilizzati controllori simulati e collisioni approssimate. Un passo in avanti sarebbe quello di studiare il passaggio da questo problema giocattolo ad un problema reale, utilizzando controllori reali e determinando le collisioni in maniera effettiva.

3. **Ricerca dell'architettura neurale e degli iperparametri.** In questo progetto si sono scelti manualmente sia l'architettura della rete neurale sia gli iperparametri, facendo brevi test su alcuni candidati. Una possibile miglioria potrebbe essere quella di utilizzare una ricerca automatica per determinarli, ad esempio tramite un algoritmo genetico.

Ringraziamenti

Ringrazio i miei relatori, Alessandro Farinelli ed Enrico Marchesini, per tutto il supporto, non solo durante questo progetto, ma anche per l'aiuto che mi hanno dato per ciò che concerne i miei studi futuri.
Ringrazio inoltre tutta la mia famiglia per avermi sostenuto durante questo percorso e per avermi motivato ad intraprendere questo cammino.

References

[1] H. V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

[2] Enrico Marchesini, Davide Corsi, Andrea Benfatti, Alessandro Farinelli, and Paolo Fiorini. Double deep q-network for trajectory generation of a commercial 7dof redundant manipulator. In *3rd IEEE International Conference on Robotic Computing, IRC 2019, Naples, Italy, February 25-27, 2019*, pages 421–422. IEEE, 2019.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[4] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[5] Thomas Simonini. Improvements in deep q learning: Dueling double dqn, prioritized experience replay, and fixed... https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/. Accessed: 2019-06-20.

[6] R. S. Sutton and A. G. Barto. *Reinforcement learning: an introduction*, chapter 1, pages 3–5. Adaptive Computation and Machine Learning. The MIT Press, 1998.

[7] R. S. Sutton and A. G. Barto. *Reinforcement learning: an introduction*, chapter 9, pages 227–233. Adaptive Computation and Machine Learning. The MIT Press, 1998.

[8] R. S. Sutton and A. G. Barto. *Reinforcement learning: an introduction*, chapter 6, pages 148–149. Adaptive Computation and Machine Learning. The MIT Press, 1998.

[9] Lei Tai and Ming Liu. Deep-learning in mobile robotics - from perception to control systems: A survey on why and why not. *CoRR*, abs/1612.07139, 2016.

[10] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 2094–2100. AAAI Press, 2016.